Name: Sahib Bajwa

ID: 107553096

CSCI 3104, Algorithms **Profs. Chen & Grochow**

Problem Set 9 – Due Fri Apr 10 11:55pm **Spring 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Informal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solutions:

- All submissions must be typed.
- You should submit your work through the **class Canvas page** only.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please allot at least as many pages per problem (or subproblem) as are allotted in this template.

Quicklinks: 1a 1b 1c 2a 2b

Name: Sahib Bajwa
ID: 107553096

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 9 – Due Fri Apr 10 11:55pm **Spring 2020, CU-Boulder**

1. Could dynamic programming be applied to give an efficient solution to the following problems? Justify your answer in terms of the optimal substructure property on the overlapping sub-problems. Even if you think the answer is yes, you do not need to give an algorithm; the question is *not* "show us how to solve it using DP", it is "give an argument as to whether DP is a reasonable approach to try here."

   (a) List maximum.

   Input: List L of numbers.

   Output: The maximum element in the list.

   I think that DP could have been a reasonable approach to try, but there is no reason to when we can simply do a linear search on the list of items. Dividing the original list into subproblems and then solving each subproblem and storing the value seams trivial when we can simply iterate through the list and keep record of the largest value we've encountered. When we do dynamic programming, we want to store the values of subproblems so that we don't have to compute them multiple times later on. In this situation, we would be storing the largest value of each subproblem after comparing to other subproblem's largest values (in my head, this would look similar to a divide and conquer algorithm (tree structure)). We would still be doing the same amount of comparisons as if we had just done a linear search for the maximum value. Due to this, I don't think DP is a reasonable approach to try here.

(b) Rod cutting.

Input: A list of values $v_1, \ldots, v_n$ for rods of length $1, \ldots, n$, respectively.

Goal: Divide a rod of length n into pieces of lengths $\ell_1, \ldots, \ell_k$ (k can vary) to maximize the total value $\sum_{i=1}^{k} v_{\ell_i}$.

Note: While this problem is discussed on GeeksForGeeks (and elsewhere), the explanation of optimal substructure on GeeksForGeeks, while not incorrect, is not sufficient explanation to demonstrate mastery of this question.

I think that DP can be a reasonable approach to try here. DP applies when we have overlapping subproblems (Prof. Chen's notes). For this problem, our overlapping subproblems would be the different lengths the maximum value possible is with the cuts of the rod possible. Since we will be building up the optimal solution to the problem by building off the optimal solutions to the subproblems, the optimal substructure property will hold. For me, I think about this problem as if we cut the rod down to its basic form (size 1), then make it one size larger and compute the optimal value for that length and so on until we have the optimal value for most multiples of the base values. Then, we can solve the problem easily with DP. So we store those values and use them for future rods of any length n. Due to this, I think DP is a reasonable approach to try here.

(c) Graph 3-coloring.

Input: A simple, undirected graph G.

Goal: Decide whether one can assign the colors $\{R, G, B\}$ to the vertices of G in such a way that no two neighbors get the same color.

Hint: You may find the notion of critical graph useful.

I think that DP can be a reasonable approach to this problem. When the graph is a critical graph, we will be able to create the goal graph since if we were to remove any of the edges or vertices, we could reduce the amount of colors needed by 1. Since we can reduce the number of colors needed by 1 every time we remove any of the edges or vertices, it is essentially a DP problem with each step of the graph having one edge/vertex removed as a solved subproblem. If G is not a critical graph, it will be more difficult to solve the problem since two arbitrary subgraphs cannot be easily used to design a coloring for a larger graph. For a graph G, we can make it so that a subproblem is maximally independent, which means that we will be able to color the subgraph to meet the goal. Putting these maximally independent subproblems that are successfully colored together will help us solve if we can get a graph that meets the color goal. This is essentially the formation of a DP problem/solution. Due to this, I think DP is a reasonable approach to try here.

Name: Sahib Bajwa
ID: 107553096

2. Write down the recurrence for the optimal solution for each of the following problems. Justify your answer.

   (a) Social distancing gold-panning. Imagine a river network in which your team can pan for gold, but no two of you can stand in adjacent positions. You have some idea of the expected amount $w(v)$ of gold you will find at each location v, but must decide in which locations your team should look.

   Input: A rooted tree T, with root vertex $r \in V(T)$, and vertex weights $w \colon V(T) \to \mathbb{R}_{\geq 0}$

   Output: A subset of vertices $P \subseteq V(T)$ such that no two vertices in P are adjacent, and maximizing the value $\sum_{v \in P} w(v)$.

   Case 1: Gold(v, w) does not select location v.
   Gold(v, w) selects best of all open positions excluding v.

   Case 2: Gold(v, w) selects location v.
   Collect expected amount $w(v)$ of gold.
   Remove adjacent positions (presumably $v - 1$ and $v + 1$ if the tree is structured that way).
   Gold(v, w) selects best of all vertexes excluding v and adjacent positions.

   Case 3: No more locations left.
   We are done (can add 0 to sum).

(b) Counting Knapsack. Here, we are considering the knapsack problem, but rather than returning the value of the optimum knapsack, or the optimum knapsack set, we are asking for the *number* of different optimum knapsacks (which all therefore have the same value).

Input: A list $L = [(w_1, v_1), \ldots, (w_n, v_n)]$, and a threshold weight W.

Output: The count of max-value knapsacks. A max-value knapsack is a subset $S \subseteq \{1, \ldots, n\}$ such that (1) $\sum_{i \in S} w_i \leq W$, and (2) the value $\sum_{i \in S} v_i$ is maximum among all subsets satisfying (1). The output should be *how many* different optimal solutions S there are.

For instance, if $L = [(1,1), (1,2), (2,2), (2,3)]$ and $W = 2$, then the output would be 2, because there are two optimal solutions: taking either the first two items or the very last item results in a valid knapsack of value 3. (Note that this is *not* just the total number of valid knapsacks; in this case there is a third set that fits within the weight threshold, namely the singleton $\{(2,2)\}$, but that set does not have optimal value.)

Describe how to use the lookup table for the Knapsack problem from class to count the number of optimal solutions.

To use the table, we first start at the position farthest to the right and farthest to the bottom of the table. We check to see if the value above this one is the same or not. If it is the same, we move to that position and do the same check. If the value is not the same and we have not hit the weight limit, then we move to the index that contains the first weight for the largest value item of the row above. We then go back to the first check. If we have hit the weight limit, then we go to the zero of the row above and finish by exiting through the remaining above zeroes.

Here is an example of this situation from class: